NS32CG16 Software Emulations

Series 32000
Preliminary
NS32CG16 Software Emulations
06/15/87 16:00

In this document, the full complement of NS32CG16 new instructions are detailed, and emulated. The emulations are exact, so far as to produce the same results as the microcode should, with the same input parameters.

Some differences should be noted, however.

Some instructions (like TBITS) use immediate values encoded into the instruction for some parameters. Where appropriate, the 32000 emulation code uses two (or more) routines to simulate this. For example, the TBITS S instruction is simulated with a BSR TBITSS.

As well, the PSR register values for certain instructions may be altered by the emulation code, but not by the microcode. The exit conditions documented for the instructions are accurate, however.

In all, these emulations should be treated as Functional Equivalents for the microcode. All input values in registers, and output values in registers and memory should be the same for both the microcode and the 32000 code.

The 32032 timings were done on a 0 wait state 32032 system, with 2 megabytes of DRAM. The 32016 timings were done on a 0 wait state 32016 system with a MMU, and 2 megabytes of DRAM.

All NS32CG16 instruction timings are ESTIMATED.

Copyright 1987
National Semiconductor
Electronic Imaging Group

BITWT Emulation code

The BITWT instruction allows a very fast logical OR-with-shift. This instruction can be used as the inner loop of a BITBLT s OR d operation, with no masks.

BITWT (Bit Aligned Word Transfer)

R0 Source pointer
R1 Destination pointer
R2 Shift value (If > 32, source becomes zero)
(If > 16, some source bits are lost)

Timing (shift of 8)

NS32CG16 Software emulation
28 clocks 96 clocks

32000 code

```
        movd        r3,tos                ;save a register for temp.
        movzwd        0(r0),r3          ;get the source value
        addqd          2,r0                ;increment pointer
        lshd          r2,r3                ;shift the source
        ord          r3,0(r1)          ;or with the destination
        addqd          2,r1                ;increment destination pointer
        movd          tos,r3                ;get back our register
```

Note that the movd r3,tos and movd tos,r3 instructions are used to obtain a working register for the algorithm. In microcode, this is not required. As well, if a number of BITWT instructions are 'in line', as is normally the case, the saving of r3 can be done at the start of the group. The restore can be done at the bottom of the group. This prevents the extra overhead between each instruction simulation.

BitBlt Series

The BITBLT (Bit Aligned Block Transfer) group of instructions require a preprocessor to obtain the masks and shift values from the more common inputs (i.e. source address/bit offset, destination address/bit offset). Below is an example of a preprocessor for the internal BitBlt instructions.

```
        .data
        .align        4
masks:        .word        0xffff,0xfffe,0xfffc,0xfff8,0xfff0
        .word                0xffe0,0xffc0,0xff80,0xff00
        .word                0xfe00,0xfc00,0xf800,0xf000
        .word                0xe000,0xc000,0x8000,0x0000

        .text

#        BitBlt preprocessor
#        Inputs:
#                r0 = Source byte address
#                r1 = Destination byte address
#                r2 = width of image in bits
#                r3 = height of image in bits
#                r4 = source bit offset
#                r5 = destination bit offset
#                r6 = source warp in bytes
#                r7 = destination warp in bytes
#        Outputs:
#                r0 = source address
#                r1 = destination address
```

```
#               r2 = shift value
#               r3 = height in lines (word quantity)
#               r4 = first mask
#               r5 = second mask
#               r6 = adjusted source warp
#               r7 = adjusted destination warp
#             0(sp)= width in words (word quantity)

pre:        movd        r2,tos               #save current width on stack
        movd        r3,tos                #save height on stack, free r3

        movd        r4,r3                #make a copy of starting bit off
        ashd        $-4,r4                #divide source bit offset by 16
        addd        r4,r0                #add word offset to source address
        addd        r4,r0

        addr        15,r2                #we only want last 4 bits of
        andd        r5,r2                #the destination x value

        ashd        $-4,r5                #divide by 16 to get word offset
        addd        r5,r1                #add offset to destination addr.
        addd        r5,r1

        movd        r3,r5                #get source bit offset
        addd        4(sp),r5        #add in width of image
        addqd        -1,r5                #and subtract one
        andd        $15,r5                #mask all but 4 lsb's
        comw        masks+2[r5:w],r5#and get rmask value

        addr        15,r4                #store mask for index into lmasks
        andd        r3,r4                #get 4 lsb's of src bit position
        subd        r4,r2                #subtract src offset from dst to get
                                #shift amount
        addr        16,r3                #calculate start bit
        subd        r4,r3                #hold start bit (16-0) in r3
        movw        masks[r4:w],r4        #get lmask value

        movd        r2,tos                #get a free register
        movd        8(sp),r2        #get width
        cmpd        r3,r2                #is w < start bits?
        ble        pre1                #no, multi-word blit, else
        andd        r5,r4                #merge the two masks
        movqd        1,r2                #and set the width to one
        br        pre1a

        .align        4
pre1:        subd        r3,r2                #width = (wbits - startbit -
1)/16 +2
        addqd        -1,r2
        ashd        $-4,r2                #width /= 16
        addqd        2,r2                #and add two

pre1a:        movd        r2,8(sp)        #store width away
```

```
        addqd   -1,r2           #adj.warp = (warp - (width - 1) * 2)
        subd        r2,r7               #and subtract width from dwarp
        subd        r2,r7               #(width is in words, warp in bytes)
        subd        r2,r6               #and swarp
        subd        r2,r6               #see above
        movd        tos,r2               #get back our shift amount

        cmpqd        0,r2               #is shift negative?
        blt        pre2a               #yes, it is negative
pre2:        addd        $16,r2               #we need a bit more shift
        addqd        -2,r1               #and to back up destination

        .align        4
pre2a:       movd        tos,r3               #get back height
#
#        Execute BitBlt function here
#
        cmpqd        0,tos                       #unstack width
        ret        0                       #and return
```

This code requires 400-500 clocks, depending on input values and processor.

This preprocessor assumes that the direction is always visual left to right and top to bottom (increasing bit/byte direction). It correctly identifies the cases where the destination pointer needs to be predecremented, and handles the special one-word-wide case.

This code sets up all data for the BBFOR, BBOR and BBXOR functions. Two additional instructions are required for the BBAND function, and may be place following the label 'pre2a' in the listing.

```
          comw      r4,r4          # Complement first mask
          comw      r5,r5          # Complement second mask
```

This correctly sets up the masks for the BBAND function, in 22 clocks.

Since the preprocessor is much the same for all internal BitBlt functions, one subroutine will serve all. A CASE instruction may be used to jump to the correct routine. Alternatively, separate preprocessor may be used for each BitBlt function for maximum speed. Using a pre-calculated address, the
CASE instruction adds 29 clocks.

The NS32CG16 uses a technique called 'Source Masking', and thus the

preprocessor may be unfamiliar to those used to working with more conventional BitBlt software/hardware.

The NS32CG16 Application notes will feature a more general purpose BitBlt preprocessor, capable of automatic determination of direction.

External BitBlt Preprocessor

Since the EXTBLT instruction communicates with the DP8510/DP8511, a different preprocessor is required. The DP8510/DP8511 use the technique called 'Destination Masking', and the masks are therefore calculated differently. Below is an example of the EXTBLT preprocessor.

```
#        External BitBlt preprocessor
#        Inputs:
#                r0 = Source byte address
#                r1 = Destination byte address
#                r2 = width of image in bits
#                r3 = height of image in bits
#                r4 = source bit offset
#                r5 = destination bit offset
#                r6 = source warp in bytes
#                r7 = destination warp in bytes
#        Outputs:
#                r0 = source address
#                r1 = destination address
#                r2 = current preload indicator
#                r3 = current width
#                r4 = actual preload indicator
#                r5 = actual width
#                r6 = adjusted source warp
#                r7 = adjusted destination warp
#              0(sp)= height in lines (word quantity)

epre:        movd        r3,tos                #save height on stack, free r3
        movd        r2,tos                 #save width on stack, free r2

        movd        r4,r3                 #make a copy of starting bit off
        ashd        $-4,r4                 #divide source bit offset by 16
        addd        r4,r0                 #add word offset to source address
        addd        r4,r0

        addr        15,r2                 #we only want last 4 bits of
        andd        r2,r3                 #the source x value, and
        andd        r5,r2                 #the destination x value
```

```
        subd        r3,r2               #shift = (dx - sx)
        andd        $15,r2               #mask back to 0-15
        movqd        0,r4               #assume no preload
        cmpqw        0,r2               #was there any shift?
        beq        epre0:b               #no, skip it.
        cmpw         r2,r3              #is the shift less than the bit off?
        bgt        epre0:b               #no... skip it
        addqd        -2,r6               #pre-decrement source warp to show
        movqd        1,r4               #preload in effect

epre0:        lshd        $8,r2                #and shift it left 8 bits to
                                #align with control register

        movd        r5,r3               #make a copy of destination x
        ashd        $-4,r5               #divide by 16 to get word offset
        addd        r5,r1               #add offset to destination addr.

        addd        r5,r1
        movd        r3,r5               #get back destination x
        addd        0(sp),r3        #add in width of image
        addqd        -1,r3                #and subtract one
        andd        $15,r3               #mask all but 4 lsb's
        movb        r3,r2               #move into position for control reg.

        andd        $15,r5               #mask to get starting x value
        movd        r5,r3               #save this value for starting bit
        lshd        $4,r5               #shift to align with control reg.
        orw        r5,r2               #and or it in.
        addr        16,r5               #calculate start bit
        subd        r3,r5               #hold start bit (16-0) in r5

        movd        0(sp),r3        #get width
        cmpd        r5,r3               #is w < start bits?
        ble        epre1:b               #no, multi-word blit, else
        movqd        1,r3               #set the width to one
        br        epre1a:b

        .align        4
epre1:        subd        r5,r3                #width = (wbits - startbit -
1)/16 +2
        addqd        -1,r3
        ashd        $-4,r3               #width /= 16
        addqd        2,r3               #and add two

epre1a:
        subd        r3,r7               #and subtract width from dwarp
        subd        r3,r7               #(width is in words, warp in bytes)
        subd        r3,r6               #and swarp
        subd        r3,r6               #see above


        .align        4
epre2a:        cmpqd        0,tos                #unstack width (now in r3)
```

```
        movd        r3,r5               #save width in r5

epre3:        movd        r4,r2               #get preload indicator
        movd        r5,r3               #get width in words
#
#      EXTBLT instruction goes here
#
        addd        r6,r0               #add source warp
        addd        r7,r1               #add destination warp
        acbw        -1,tos,epre3       #and loop until done
        cmpqd        0,tos               #unstack height
        ret        0
```

This code requires about 400 clocks to do the preprocessing, and about 41 clocks per line of overhead for the loop starting at the label 'epre3:'. This can be unrolled, if the height is constant.

BitBlt Emulation

The four primary BitBlt instructions (BBOR, BBAND, BBFOR and BBXOR) are
somewhat difficult to emulate, since some of the the values are immediate (such as IS). Two methods of emulation are therefore presented. The first is a general purpose BITBLT. Along with the width at 0(sp), the general purpose BITBLT expects a function word at 2(sp) prior to the BSR to enter this function. The function word is defined in the listing below. Note that 0(sp) becomes 4(sp) due to the BSR to enter this function.

```
#Emulations of BBOR BBAND BBXOR and BBFOR
#
#        Inputs:
#                r0 = source address
#                r1 = destination address
#                r2 = shift value
#                r3 = height in lines (word quantity)
#                r4 = first mask
#                r5 = second mask
#                r6 = source warp
#                r7 = destination warp
#              4(sp)= width in words (word quantity)
#
#              6(sp)= Function/direction (immediate in microcode)
#                     bit 1 0
#                         0 0        AND
#                         0 1        OR
```

```
#                         1 0        XOR
#                         1 1        FOR
#                       bit 2 = NOT source
#                       bit 3 = RD (Reverse Direction)
#
#        Ouputs:
#                all registers destroyed.
#                Data in memory changed.
#
        .align        4
bbgen:        movd        r3,tos                #save height, free r3
        movd        r6,tos                #save source warp, free r6
        movd        r7,tos                #save destination warp, free r7
        movqd        3,r7                #place mask in r7
        andd        18(sp),r7        #get function code
        addr        r7[r7:q],r7        #r7 *= 9
        addr        andmfnc:w[r7:d],tos #figure address (each func is 36
bytes)
#
#Now register map looks like:
#
#                r0 = source address
#                r1 = destination address
#                r2 = shift value
#                r3 = temporary register for blt work
#                r4 = first mask
#                r5 = second mask
#                r6 = horizontal increment
#                r7 = current width
#             21(sp)= width in words (word quantity)
#
#             23(sp)= Function/direction (immediate in microcode)
#
#              0(sp)= flags via SPRB upsr
#              1(sp)= src to destination routine address
#              5(sp)= destination warp
#              9(sp)= source warp
#             13(sp)= height        (word quantity)
#

        movb        22(sp),r7        #get function byte

        tbitb        2,r7                #set flags for inversion of source
        cmpqb        0,0                #assume no inversion
        bfc        bbg1a:b                #yes, no inversion
        cmpqb        1,0                #please invert source
bbg1a:        sprb        upsr,tos        #save inversion status in the Z
flag

        movqd        2,r6                #assume incrementing direction
        tbitb        3,r7                #is it?
        bfc        bbgen1:b        #it is.
        movqd        -2,r6                #it is decrementing direction
```

```
bbgen1:
        movw        21(sp),r7        #get current width
bbgen1a:
        lprb        upsr,0(sp)        #get back Z flag
        jsr        0(1(sp))        #do function
        addqd        -2,r7                #width -= 2
        cmpqd        0,r7                #if width is now zero, do other mask
        beq        bbgenmsk2:b        #yes, do other mask
        bgt        bbgenend:b        #if width is < zero, line is done

        lprb        upsr,0(sp)        #get back Z flag
        .align        4
bbgen2:        jsr        14(1(sp))        #do function
        acbd        -1,r7,bbgen2        #and loop for all inner words

bbgenmsk2:
        lprb        upsr,0(sp)        #get back Z flag
        movd        r4,r7                #copy left mask to temp
        movd        r5,r4                #use right mask now
        jsr        0(1(sp))        #do function
        movd        r7,r4                #get back left mask
bbgenend:
        addd        9(sp),r0        #add source warp
        addd        5(sp),r1        #add destination warp
        acbw        -1,13(sp),bbgen1#and loop for entire blt

        adjspb        $-17                #unstack all items
        ret        0                #and return
```

```
        .align        4
andmfnc:
        movqd        -1,r3                #fill upper bits with 1's
        movw        0(r0),r3        #get word
        beq        andf1:b                #if no inversion, skip
        comw        r3,r3                #inversion of source
andf1:        orw        r4,r3                #do mask
        br        andf2:b                #rejoin code

andfunc:
        movqd        -1,r3                #fill upper bits with 1's
        movw        0(r0),r3        #get word
        beq        andf2:b                #if no inversion, skip
        comw        r3,r3                #inversion of source
andf2:        rotd        r2,r3                #rotate to align
        andd        r3,0(r1)        #and to destination
        addd        r6,r0                #add in horizontal increment
        addd        r6,r1                #to source and destination
        ret        0                #and return


        .align        4
ormfnc:        movzwd        0(r0),r3        #get word, fill upper bits with
```

```
        beq        orf1:b                #if no inversion, skip
        comw        r3,r3                #inversion of source
orf1:        andw        r4,r3                #do mask
        br        orf2:b               #rejoin code
        nop                          #make length 14 bytes

orfunc:        movzwd        0(r0),r3        #get word, fill upper bits with
0
        beq        orf2:b                #if no inversion, skip
        comw        r3,r3                #inversion of source
orf2:        rotd        r2,r3                #rotate to align
        ord        r3,0(r1)        #or to destination
        addd        r6,r0                #add in horizontal increment
        addd        r6,r1                #to source and destination
        ret        0                #and return

        .align        4
xormfnc:movzwd        0(r0),r3        #get word, fill upper bits with 0
        beq        xorf1:b                #if no inversion, skip
        comw        r3,r3                #inversion of source
xorf1:        andw        r4,r3                #do mask
        br        xorf2:b                #rejoin code
        nop                          #make length 14 bytes

xorfunc:movzwd        0(r0),r3        #get word, fill upper bits with 0
        beq        xorf2:b                #if no inversion, skip
        comw        r3,r3                #inversion of source
xorf2:        rotd        r2,r3                #rotate to align
        xord        r3,0(r1)        #xor to destination
        addd        r6,r0                #add in horizontal increment
        addd        r6,r1                #to source and destination
        ret        0                #and return


        .align        4
frmfnc:        movzwd        0(r0),r3        #get word, fill upper bits with
0
        andw        r4,r3                #do mask
        br        frf2:b               #rejoin code
        nop                          #make length 14 bytes
        nop
        nop
        nop
        nop
        nop

frfunc:        movzwd        0(r0),r3        #get word, fill upper bits with
0
frf2:        rotd        r2,r3                #rotate to align
        ord        r3,0(r1)        #or to destination
        addqd        2,r0                #add in horizontal increment
        addqd        2,r1                #to source and destination
```

```
        ret        0                #and return
```

The general purpose case can be unrolled, and placed in a number of specific case routines. This helps to show the functionality of each case. Below is the BBOR emulation.

```
#Emulations of BBOR
#
#        Inputs:
#                r0 = source address
#                r1 = destination address
#                r2 = shift value
#                r3 = height in lines (word quantity)
#                r4 = first mask
#                r5 = second mask
#                r6 = source warp
#                r7 = destination warp
#              0(sp)= width in words (word quantity)
#
#              2(sp)= Function/direction (immediate in microcode)
#                     bit 0 = NOT source
#                     bit 1 = RD (Reverse Direction)
#
#        Ouputs:
#                all registers destroyed.
#                Data in memory changed.
#

bbor:
        movd        r3,tos               #save height, free r3
        movd        r6,tos               #save source warp, free r6
        movd        r7,tos               #save destination warp, free r7

#
#Now register map looks like:
#
#                r0 = source address
#                r1 = destination address
#                r2 = shift value
#                r3 = temporary register for blt work
#                r4 = first mask
#                r5 = second mask
#                r6 = horizontal increment
#                r7 = current width
#             12(sp)= width in words (word quantity)
#
#             14(sp)= Function/direction (immediate in microcode)
#
#              0(sp)= destination warp
```

```
#               4(sp)= source warp
#               8(sp)= height        (word quantity)
#
        movqd        2,r6               #assume incrementing direction
        tbitb        1,14(sp)        #is it?
        bfc        bbor1:b               #it is.
        movqd        -2,r6               #it is decrementing direction
bbor1:        movw        12(sp),r7        #get current width

        movzwd        0(r0),r3        #get first word in line
        tbitb        0,14(sp)        #do invert source?
        bfc        bbor1a:b        #no, no inversion
        comw        r3,r3               #else invert the source
bbor1a:        andw        r4,r3               #mask the source
        rotd        r2,r3               #shift left to align
        ord        r3,0(r1)        #or to destination
        addd        r6,r0               #add in horizontal increment
        addd        r6,r1               #to source and destination
        addqd        -2,r7               #width -= 2
        cmpqd        0,r7               #if width is now zero, do other mask
        beq        bbormsk2:b        #yes, do other mask
        bgt        bborend:w        #if width is < zero, line is done

        tbitb        0,14(sp)        #set flags for inversion of source
        cmpqb        0,0               #assume no inversion
        bfc        bbor2:b               #yes, no inversion
        cmpqb        1,0               #please invert source

        .align        4
bbor2:        movzwd        0(r0),r3        #get next word
        beq        bbor2a:b        #if Z flag set, no inversion
        comw        r3,r3
bbor2a:        rotd        r2,r3               #shift to align
        ord        r3,0(r1)        #and or to destination
        addd        r6,r0               #add in horizontal increment
        addd        r6,r1               #to source and destination
        acbd        -1,r7,bbor2        #and loop for all inner words

bbormsk2:
        movzwd        0(r0),r3        #get last word
        tbitb        0,14(sp)        #do invert source?
        bfc        bbor3a:b        #no, no inversion
        comw        r3,r3               #else invert the source
bbor3a:        andw        r5,r3               #do second mask
        lshd        r2,r3               #shift to align
        ord        r3,0(r1)        #or to destination
        addd        r6,r0               #add horizontal increment
        addd        r6,r1               #to source and destination

bborend:addd        4(sp),r0        #add source warp
        addd        0(sp),r1        #add destination warp
        acbw        -1,8(sp),bbor1        #and loop for entire blt
```

```
        adjspb        $-12                #unstack all items
        ret        0                 #and return
```

External BitBlt

With the external BITBLT, exact emulation is impossible, since the hardware signals do not exist on the 32000 series. Simulation via a hardware register is possible, however, and is reproduced below.

```
#Emulations of EXTBLT
#
#        Inputs:
#                r0 = source address
#                r1 = destination address
#                r2 = number of source reads
#                r3 = width in words
#
#        Ouputs:
#                r0, r1 set to the next transfer address
#                r3 is modified. r2 is altered
#                Data in memory changed.
#
        .data
        .align        4
masks:        .word        0                #mask enables
nme:        equ        0                #no masks on
lme:        equ        1                #left mask on
rme:        equ        2                #right mask on

        .text
extblt:
        tbitb        1,r2                #do we increment or decrement
        bfs        ext2:b                #we decrement
        tbitb        0,r2                #two source reads?
        bfc        ext1a:b                #no, just one
        movw        0(r0),r2        #do a pre-read
        addqd        2,r0                #increment source address
ext1a:        movqw        lme,masks        #turn on left mask
        addqd        -2,r3                #how many left?
        cmpqd        0,r3                #was it a one word wide?
        bgt        ext1b:b                #yes, do right mask too
        orw        0(r0),0(r1)        #do read/read-modify-write
        addqd        2,r0                #increment source address
        addqd        2,r1                #increment destination address
        beq        ext1c:b                #it was two words wide
        movqw        nme,masks        #no masks now
ext1d:        orw        0(r0),0(r1)        #do read/read-modify-write
        addqd        2,r0                #increment source address
```

```
        addqd        2,r1                #increment destination address
        acbd        -1,r3,ext1a        #and loop until done
ext1c:        movqw        rme,masks        #turn on right mask
ext1e:        orw        0(r0),0(r1)        #do last read/read-modify-write
        addqd        2,r0                #increment source address
        addqd        2,r1                #increment destination address
        ret        0                #we are done


ext1b:        movqw        lme+rme,masks        #turn on both masks
        br        ext1e                #and do the read/read-modify-write

ext2:        tbitb        0,r2                #two source reads?
        bfc        ext2a:b                #no, just one
        movw        0(r0),r2        #do a pre-read
        addqd        -2,r0                #increment source address
ext2a:        movqw        lme,masks        #turn on left mask
        addqd        -2,r3                #how many left?
        cmpqd        0,r3                #was it a one word wide?
        bgt        ext2b:b                #yes, do right mask too
        orw        0(r0),0(r1)        #do read/read-modify-write
        addqd        -2,r0                #increment source address
        addqd        -2,r1                #increment destination address
        beq        ext2c:b                #it was two words wide
        movqw        nme,masks        #no masks now
ext2d:        orw        0(r0),0(r1)        #do read/read-modify-write
        addqd        -2,r0                #increment source address
        addqd        -2,r1                #increment destination address
        acbd        -1,r3,ext2a        #and loop until done
ext2c:        movqw        rme,masks        #turn on right mask
ext2e:        orw        0(r0),0(r1)        #do last read/read-modify-write
        addqd        -2,r0                #increment source address
        addqd        -2,r1                #increment destination address
        ret        0                #we are done

ext2b:        movqw        lme+rme,masks        #turn on both masks
        br        ext2e                #and do the read/read-modify-write
```

MOVMP Emulation code

Emulating the MOVMP instruction in 32000 code requires three separate routines, due to the fact the the MOVMP instruction can be used with BYTE, WORD or DOUBLE source/destinations. The timing for all, along with the 32000 emulations, follows.

MOVMPi (Move Multiple Pattern)

R0 Destination pointer
R1 Pointer increment (signed, 32 bit)

R2 Number of elements
R3 Source pattern

In this instruction **R0, the destination pointer, and R2, the number of elements, are altered**. R1, and R3 are preserved.

Timing

In the following table, N represents the number of elements (R2) supplied to the MOVMP instruction.

NS32CG16 Software emulation
Byte 11 * N 14 + 43 * N
Word 11 * N 14 + 43 * N
Double 11 * N 14 + 47 * N

The actual 32000 code used is reproduced on the following page.

32000 code

```
movmpb:
        cmpqd        0,r2               ;any to move?
        beq        lp2                ;no, skip it
        .align        4
lp1:        movb        r3,0(r0)        ;stuff a byte out now
        addd        r1,r0               ;increment address
        acbd        -1,r2,lp1        ;and back for more
lp2:

     - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

movmpw:
        cmpqd        0,r2               ;any to move?
        beq        lp4                ;no, skip it
        .align        4
lp3:        movw        r3,0(r0)        ;stuff a word out now
        addd        r1,r0               ;increment address
        acbd        -1,r2,lp3        ;and back for more
lp4:


     - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
movmpd:
        cmpqd        0,r2               ;any to move?
        beq        lp4                ;no, skip it
        .align       4
lp3:        movd        r3,0(r0)        ;stuff a double out now
        addd        r1,r0               ;increment address
        acbd        -1,r2,lp3        ;and back for more
lp4:
```

SBITS instruction definition

The SBITS instruction is typically used to set a number of bits in a block of memory to '1'. In order to make this instruction as fast as possible, it uses a 1024 byte external table. This table is defined at the end of this document. The formal definition of the SBITS instruction is:

SBITS (Set Bit String)

R0 Base address
R1 Bit offset
R2 Run length
R3 Address of SBITS table

Note that SBITS may safely be called with a run length of zero. This simplifies boundary conditions. SBITS may only be used in the horizontal (bit offset increasing) direction. The SBITPS instruction can be used with a register supplying the bit increment value, allowing perpendicular lines to be drawn as well.

SBITS first compares the run length with 25. If the length is greater than 25, SBITS sets the F flag and exits. This allows the external software to use a more efficient algorithm to set large quantities of bits. If your application only requires 1-24 bits, no external software is required to use this instruction. if the run length is less than or equal to 24, SBITS clears the F flag.

SBITS then calculates the starting base address from the base address supplied, and the bit offset, according to the following formula.

base address = base address + (bit offset DIV 8)
bit offset = (bit offset MOD 8)

The actual 32000 code representation of this is:

```
movd  r1,temp       ;save the bit offset for a second
ashd  -3,r1         ;divide the bit offset by 8, signed
addd  r1,r0         ;r0 = r0 + (r3 / 8)
movqd 7,r1          ;place a mask in r1
andd  temp,r1       ;r1 = (bit offset .and. 7)
```

This adjusts the base pointer to the nearest byte of the data to be affected. The bit offset will always, therefore, be in the range 0 to 7.

Assuming the run length is 24 or less, SBITS then indexes into the table according to the following equation:

index = table address + 4 * (run length + (32 * bit offset))

Since the bit offset has been previously adjusted to the range 0 to 7, only a 1024 byte table is required (8 rows of 128 bytes).

The actual 32000 code representation of this would be:

```
movd  r1,temp            ;store bit offset to temp
lshd  5,temp             ; temp = (32 * bit offset)
addd  r2,temp            ;temp = temp + run length
lshd  2,temp             ;temp = temp * 4
addd  r3,temp            ;index = base of table + offset
```

SBITS then does a logical OR to move the specified bits into position, from the table to the destination address.

The 32000 code representation of this would be:

```
ORD  0(temp),0(r0)       ;Set the required bits
```

Again, note that while no special test is made for a run length of zero, no harm will occur to the destination data since the table contains all zeros at a run length of 0.

If the run length is greater than 24 bits, an external software algorithm should be used. This increases the speed even more. A sample software algorithm incorporating the MOVMP instruction is reproduced on the following page.

```
        .align        4
bigset:        save        [r0,r1,r2,r3,r4,r5,r6]        ;save registers we
will affect
        movd        r1,r4                ;get current bit offset
        ashd        -3,r4                ;divide by eight to get byte offset
        addd        r4,r0                ;add in base. r0 is new base pointer
        andd        7,r1                ;mask off msb's of bit pointer to
                                ;get bit = bit offset mod 8
;Now we have true base address and bit offset within base. Now move
;to double word alignment. This speeds up the MOVMPD.

        movqd        3,r4                ;place mask in r5
        andd        r0,r4                ;get low three bits of address
        xord        3,r4                ;and get bytes left to alignment
        addqd        1,r4                ;rem += 1 (for the byte we are on)
        ashd        3,r4                ;rem *= 8 to get bits to alignment
        subd        r1,r4                ;subtract current bit offset
        cmpd        r4,r2                ;is this more than bits left?
        bge        shrt:w                ;it is, do it the short way

        cmpd        32,r4                ;if we are double aligned, go
        beq        mvm:b           ;do the MOVMPD

        movd        r1,r5                ;calculate index into table
        lshd        5,r5                ;index = 32 * bit offset
        addd        r4,r6                ;index += run length
        ord        0(r4)[r6:d],0(r0);or in required bits
        andd        h'fffffffc,r0        ;mask address before increment
        addqd        4,r0                ;bump to next double
        subd        r4,r2                ;zap sp'd bits off
mvm:        movd        r2,r4                ;save run length for a minute
        movd        r3,r5                ;and save pointer to table
        ashd        -5,r2                ;r1 = r1 / 32 = number of doubles
        movd        1020(r3),r3        ;get source pattern from table
        movqd        4,r1                ;increment is r1
        MOVMPD                        ;move multiple pattern instruction
        andd        h'1f,r4                ;mask off all but last 32 bits
        ord        0(r5)[r4:d],0(r0);insert the last few bits
        restore        [r0,r1,r2,r3,r4,r5,r6];        restore saved registers
        ret        0                ;and return
```

```
        .align        4
shrt:
        movd        r1,r4                       ;calculate index into table
        lshd        5,r4                       ;index = 32 * bit offset
        addd        r2,r4                       ;index += run length
        ord        0(r3)[r4:d],0(r0)        ;or in required bits
        restore        [r0,r1,r2,r3,r4,r5,r6]        ;restore saved registers
        ret        0
```

Other uses of SBITS

The SBITS instruction may be used for other purposes such as a simple line draw/fill. By changing the table to a pattern other than all ones, SBITS can perform a very fast bit aligned pattern fill. The pattern width is 32 bits, and the height is one scan line. Multiple tables may be used to create any height, and tables can be created on the fly. To create your pattern, simply place the desired 32 bit pattern in R3, point R1 to the existing SBITS table, and point R2 to a 1024 byte area of memory. The following code will create a new pattern table:

```
crepat:
         addrd @16,r0           ;we need to mask 256 doubles
        .align 4                ;This code has been unrolled
crlp:        movd         0(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,0(r2)        ;store in destination
        movd         4(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,4(r2)        ;store in destination
        movd         8(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,8(r2)        ;store in destination
        movd         12(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,12(r2)        ;store in destination
        movd         16(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,16(r2)        ;store in destination
        movd         20(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,20(r2)        ;store in destination
        movd         24(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,24(r2)        ;store in destination
        movd         28(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,28(r2)        ;store in destination
        movd         32(r1),r4        ;get current value
```

```
        andd        r3,r4               ;mask it
        movd        r4,32(r2)        ;store in destination
        movd         36(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,36(r2)        ;store in destination
        movd         40(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,40(r2)        ;store in destination
        movd         44(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,44(r2)        ;store in destination
        movd         48(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,48(r2)        ;store in destination
        movd         52(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,52(r2)        ;store in destination
        movd         56(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,56(r2)        ;store in destination
        movd         60(r1),r4        ;get current value
        andd        r3,r4               ;mask it
        movd        r4,60(r2)        ;store in destination
        addd        64,r2               ;increment destination
        addd        64,r1               ;increment source
        acbb -1,r0,crlp;continue for the rest
        ret 0
```

This code occupies about 150 bytes, and takes 9,127 clocks to create a new pattern. At 15 Mhz, that is about .6 msec.

Character expansion

A simple Run Length Limited code may be used in conjunction with SBITS to
decode fonts from a small amount of space. A 32 by 32 bit font, for example, can be compressed about 50 % (from 128 bytes to 60 bytes, in this example) per character. A sample 32000 program is reproduced on the following page, along with the encoded character table for the '!' character (ASCII 0x21). The routine executes in 1-2 msec, depending on input data.

```
; Character expansion routine.
;        Inputs:
;                r0 = destination base address (expanded data)
```

```
;                r1 = destination bit offset
;                r2 = used internally for run length
;                r3 = sbits table base
;                r4 = source base address      (compressed data)
;                r5 = source bit offset
;        Constants (May be memory-based or equated):
;                xwarp = destination memory width in bytes
;                cwidth = character width in bits
;                cheight = character height in bits
;        Outputs:
;                all registers preserved
;                bits pointed to by r0 on entry modified
;Notes:
;        This routine uses a five bit RLL code with embedded
;        zero/1 information. This simple technique yields a modest
;        50% compression ratio (from 128 bytes for a ! to 60 bytes).
;        Other techniques may be used.
cwidth:        equ        32                ;a 32 by 32 character
cheight:equ        32
xwarp:        equ        4                ;memory is 4 bytes 'wide'
char:        save        [r0,r1,r2,r3,r4,r5,r6,r7]
        movd        r1,r6                ;calculate field end position
        addd        cwidth,r6        ;add width to start bit
        movd        cheight,r7        ;do the loop cheight times
line:        movd        r1,tos                ;save bit offset
sc1:        extd        r5,0(r4),r2,5        ;get the next five bits
        addqd        5,r5                ;increment pointer for next access
        cbitb   4,r2            ;test and clear bit 4 of R2
        bfc        whitesp                ;if clear, we wanted white space
        SBITS                        ;else set the bits we need
whitesp:                        ;and join up with whitespace
        addd        r2,r1                ;add run length to current bit
offset
        cmpd        r1,r6                ;are we done with this line?
        blt        sc1                ;no, back for more
        movd    tos,r1                ;get back bit offset
        addd        xwarp,r0        ;add xwarp to starting position
        acbd        -1,r7,line        ;and go back for another line
        restore        [r0,r1,r2,r3,r4,r5,r6,r7]        ;and we are done.
        ret        0
chrtab:
        double        h'9d4709ef,h'e751cea3,h'39d473a8,h'8e751cea
        double        h'a39d473a,h'a8e751ce,h'ea39d473,h'3a8e751c
        double        h'cea39d47,h'73a8e751,h'deea39d4,h'f789ef13
        double        h'51c27bc4,h'd473a8e7,h'13deea39
```

SBITS Emulation code

The exact emulation for the SBITS instruction, in 32000 code, is reproduced below following the description.

SBITS (Set Bit String)

R0 Base address
R1 Bit offset
R2 Run length
R3 Address of SBITS table

```
; --- start of SBITS emulation ---
        cmpd        24,r2               ;are we setting more than 24 bits?
        blt        out:b               ;yes, use the bigger algorithm

        save        [r4,r5]               ;we will trash these, so save them
        movqd        7,r4               ;mask off msb's of bit offset
        andd        r1,r4               ;with an and (r4 = r2 mod 8)
        lshd        5,r4               ;index = 32 * bit offset
        addd        r2,r4               ;index += run length
        movd        r1,r5               ;get bit offset
        ashd        -3,r5               ;divide by eight to get byte offset
        ord        0(r3)[r4:d],0(r0)[r5:b]        ;or in required bits
        restore        [r4,r5]                ;restore registers
out:        ret        0               ;and return

; --- End of SBITS emulation ---
```

Estimated Timing

Run length NS32CG16 32000
1-25 bits 43 Clocks 205 Clocks
26-64 bits 389 Clocks 389 Clocks
64 + bits 525 + (11 547 + (33
clocks per clocks per
double) double)

In the last entry, '11 clocks per double' refers to the number of double words (32 bits) over and above the 64 bit initialization time. Thus, a run length of 160 bits would take an estimated 558 clocks (525 + ((160 - 64)/32 * 11) on the NS32CG16.

Tables

The table for the SBITS instruction is comprised of 8 groups of 32 double words, or 1024 bytes in all. The table may reside in code (PC relative) or

data (SB relative) space.

```
stab:
        double          h'00000000,h'00000001,h'00000003,h'00000007
        double          h'0000000f,h'0000001f,h'0000003f,h'0000007f
        double          h'000000ff,h'000001ff,h'000003ff,h'000007ff
        double          h'00000fff,h'00001fff,h'00003fff,h'00007fff
        double          h'0000ffff,h'0001ffff,h'0003ffff,h'0007ffff
        double          h'000fffff,h'001fffff,h'003fffff,h'007fffff
        double          h'00ffffff,h'01ffffff,h'03ffffff,h'07ffffff
        double          h'0fffffff,h'1fffffff,h'3fffffff,h'7fffffff
        double          h'00000000,h'00000002,h'00000006,h'0000000e
        double          h'0000001e,h'0000003e,h'0000007e,h'000000fe
        double          h'000001fe,h'000003fe,h'000007fe,h'00000ffe
        double          h'00001ffe,h'00003ffe,h'00007ffe,h'0000fffe
        double          h'0001fffe,h'0003fffe,h'0007fffe,h'000ffffe
        double          h'001ffffe,h'003ffffe,h'007ffffe,h'00ffffe
        double          h'01ffffe,h'03ffffe,h'07ffffe,h'0ffffffe
        double          h'1ffffffe,h'3ffffffe,h'7ffffffe,h'ffffffe
        double          h'00000000,h'00000004,h'0000000c,h'0000001c
        double          h'0000003c,h'0000007c,h'000000fc,h'000001fc
        double          h'000003fc,h'000007fc,h'00000ffc,h'00001ffc
        double          h'00003ffc,h'00007ffc,h'0000fffc,h'0001fffc
        double          h'0003fffc,h'0007fffc,h'000ffffc,h'001ffffc
        double          h'003ffffc,h'007ffffc,h'00ffffc,h'01ffffc
        double          h'03ffffc,h'07ffffc,h'0ffffffc,h'1ffffffc
        double          h'3ffffffc,h'7ffffffc,h'ffffffc,h'ffffffc
        double          h'00000000,h'00000008,h'00000018,h'00000038
        double          h'00000078,h'000000f8,h'000001f8,h'000003f8
        double          h'000007f8,h'00000ff8,h'00001ff8,h'00003ff8
        double          h'00007ff8,h'0000fff8,h'0001fff8,h'0003fff8
        double          h'0007fff8,h'000ffff8,h'001ffff8,h'003ffff8
        double          h'007ffff8,h'00fffff8,h'01fffff8,h'03fffff8
        double          h'07fffff8,h'0ffffff8,h'1ffffff8,h'3ffffff8
        double          h'7ffffff8,h'fffffff8,h'fffffff8,h'fffffff8
        double          h'00000000,h'00000010,h'00000030,h'00000070
        double          h'000000f0,h'000001f0,h'000003f0,h'000007f0
        double          h'00000ff0,h'00001ff0,h'00003ff0,h'00007ff0
        double          h'0000fff0,h'0001fff0,h'0003fff0,h'0007fff0
        double          h'000ffff0,h'001ffff0,h'003ffff0,h'007ffff0
        double          h'00fffff0,h'01fffff0,h'03fffff0,h'07fffff0
        double          h'0ffffff0,h'1ffffff0,h'3ffffff0,h'7ffffff0
        double          h'fffffff0,h'fffffff0,h'fffffff0,h'fffffff0
        double          h'00000000,h'00000020,h'00000060,h'000000e0
        double          h'000001e0,h'000003e0,h'000007e0,h'00000fe0
        double          h'00001fe0,h'00003fe0,h'00007fe0,h'0000ffe0
        double          h'0001ffe0,h'0003ffe0,h'0007ffe0,h'000ffffe0
        double          h'001fffe0,h'003fffe0,h'007fffe0,h'00ffffe0
        double          h'01fffffe0,h'03fffffe0,h'07fffffe0,h'0ffffffe0
        double          h'1ffffffe0,h'3ffffffe0,h'7ffffffe0,h'ffffffe0
        double          h'fffffffe0,h'fffffffe0,h'fffffffe0,h'fffffffe0
```

```
double        h'00000000,h'00000040,h'000000c0,h'000001c0
double        h'000003c0,h'000007c0,h'00000fc0,h'00001fc0
double        h'00003fc0,h'00007fc0,h'0000ffc0,h'0001ffc0
double        h'0003ffc0,h'0007ffc0,h'000fffc0,h'001fffc0
double        h'003fffc0,h'007fffc0,h'00ffffc0,h'01ffffc0
double        h'03ffffc0,h'07ffffc0,h'0fffffc0,h'1fffffc0
double        h'3fffffc0,h'7fffffc0,h'ffffffc0,h'ffffffc0
double        h'ffffffc0,h'ffffffc0,h'ffffffc0,h'ffffffc0
double        h'00000000,h'00000080,h'00000180,h'00000380
double        h'00000780,h'00000f80,h'00001f80,h'00003f80
double        h'00007f80,h'0000ff80,h'0001ff80,h'0003ff80
double        h'0007ff80,h'000fff80,h'001fff80,h'003fff80
double        h'007fff80,h'00ffff80,h'01ffff80,h'03ffff80
double        h'07ffff80,h'0fffff80,h'1fffff80,h'3fffff80
double        h'7fffff80,h'ffffff80,h'ffffff80,h'ffffff80
double        h'ffffff80,h'ffffff80,h'ffffff80,h'ffffffff
```

Set Bit Perpendicular (SBITPS)

The SBITPS instruction can be used for many purposes including:

o Drawing vertical lines
o Drawing 45 degree lines
o Drawing 90 and 270 rotated stroke fonts

Included with this document is timing of both the software emulation, and the NS32CG16 microcode.

The SBITPS instruction is defined below:

SBITPS Set Bit Perpendicular String

R0 Base address
R1 Starting bit offset
R2 Run length
R3 Space warp, signed 32 bit, bit value

All registers are preserved by this instruction. Since the space warp (R3) is signed, drawing a line 'to the left' of the current graphics cursor is permitted.

Timing

The NS32CG16 executes this instruction in 41 * N, where N is the run length

(R2). A run length of zero indicates that no bits should be set.

The 32000 code emulation of the SBITPS instruction (reproduced on the following page), executes in 79 + 66 * N, again where N is the run length (R2). The 79 clock overhead includes the save and restore time, but does not include the BSR/RET overhead.

```
; Version 1.0  Mon Mar 23 09:19:48 1987
; Exact emulation of new SBITPS instruction.
;
;        Set bit Perpendicular string emulation code
;        Inputs:
;                r0 = base address
;                r1 = bit offset
;                r2 = Run length
;                r3 = Space warp (in bits)
;        Ouputs:
;                Bits set in memory
;                stack used to save registers (8 bytes)
;               No registers changed
;
sbitps:
;        SBITPS

; --- start of SBITPS emulation ---

        cmpqd        0,r2               ;is it a zero run length?
        beq        sbend:b               ;it is, exit now
        save        [r1,r2]               ;save the registers we will change
sblp:        sbitd        r1,0(r0)        ;set the required bit
        addd        r3,r1               ;add the bit warp in
        acbd        -1,r2,sblp        ;and loop back for more
        restore        [r1,r2]                ;restore our saved registers
sbend:        ret        0               ;and return
; --- end of SBITPS emulation ---
```

TBITS Emulation code

The TBITS instruction is generally used for RLL (Run Length Limited) compression, although it can also be used to find boundaries in graphics patterns. Below are some examples of RLL compression algorithms implemented
using the TBITS instruction.

TBITS 0 | 1 (Test Bit String)

R0 Source base address
R1 Source bit offset
R2 Destination RLL code
R3 Maximum value RLL code
R4 Maximum source bit offset

R1 and R2 are modified by this instruction. All other registers are preserved.

There are two possible operations for TBITS: count zeros, and count ones. TBITS 0 scans for zeros starting at the source bit offset, and stops when it finds a one bit. TBITS 1 scans for ones, and stops when it finds a zero bit. Other exit conditions are possible, as is shown below:

Flags
Condition Z N L F Notes
R2 < R3, bit found 1 0 0 F F reflects last bit tested
R2 => R3, not found 1 0 0 F F reflects last bit tested
R1 => R4, not found 0 1 1 F F reflects last bit tested
R1 => R4 on entry 0 1 1 0/1 F = n (TBITS n)

The last entry in the table is for the case of the starting bit offset already larger than the maximum bit offset. In this case, R2 is zeroed, and the F flag is set to the condition the instruction was testing for.

32000 emulation

The 32000 emulation of this instruction is presented below.

```
; Version 1.1  Thu Mar 26 11:28:07 1987
; Exact emulation of TBITS instruction.
;
;       Test bit string emulation code
;       Inputs:
;               r0 = base address
;               r1 = starting bit offset
;               r2 = Destination run length (bits)
;               r3 = run length limit (bits)
;               r4 = max bit offset
;       Ouputs:
;               r2 = set to run length (max(r3,run length))
;               r1 = current bit offset
;               No data in memory changed
```

```
;                No other registers altered
;Notes:
;        Two routines are used to emulate the TBITS instruction.
;        tbits1 emulates TBITS 1
;        tbits0 emulates TBITS 0
;
tbits0:
;        TBITS        0

; --- start of TBITS emulation ---
        movqd        0,r2               ;set to run length of zero initially
        bicpsrb        h'20               ;set the F flag (0 not found yet)
        br        tloop:b               ;and jump into the loop

        .align        4
tbitzero:
        tbitd        r1,0(r0)        ;is the bit set?
        bfs        tboutz:b        ;it is set! exit now.
        addqd        1,r1               ;move to next bit
        addqd        1,r2               ;add one to run length
tloop:        cmpd        r1,r4               ;is the starting bit offset
already
        bge        tbout:b               ;at edge? if so, exit
        cmpd        r2,r3               ;is the run length already over max?
        blt        tbitzero        ;no, keep looking

tboutz:        cmpqb        0,0               ;set Z=1, N=0, L=0, F=F
        ret        0

tbout:        cmpqb        2,1               ;set Z=0, N=1, L=1, F=F
        ret        0               ;and we are done




;
;
;        TBITS        1
;

tbits1:
        movqd        0,r2               ;set to run length of zero initially
        bispsrb        h'20               ;clear the F flag (1 not found
yet)
        br        tloop1:b        ;and jump into the loop

        .align        4
tbitone1:
        tbitd        r1,0(r0)        ;is the bit set?
        bfc        tboutz               ;it is clear! exit now.
```

```
        addqd        1,r1                ;move to next bit
        addqd        1,r2                ;add one to run length
tloop1:        cmpd        r1,r4                ;is the starting bit offset
already
        bge        tbout                ;at edge? if so, exit
        cmpd        r2,r3                ;is the run length already over max?
        blt        tbitone1        ;no, keep looking

;        br        tboutz                ;if we fall out, exit

        cmpqb        0,0                ;set Z=1, N=0, L=0, F=F
        ret        0                ;if we fall out, we are done
; --- end of TBITS emulation ---
```

Encoding techniques

There are several encoding techniques using RLL: Alternating 0/1, embedded
control, variable length and Huffman post-processing are a few of them. Three techniques are presented on the following pages, including Alternating 0/1, embedded control, and an optimized embedded control with no run lengths of zero. The best RLL encoding technique is totally dependent on the data set involved. Each of these routines (using the 32000 emulation code) take about 22 msec to compress a 32 by 32 font.

```
;Comp1 does an alternating 0/1 compression, with a fixed 3 bit
;run length (0 to 7 as the run). The data to be compressed is
;an 32 by 32 font matrix, initialy aligned on a byte boundary (byte
;alignment is not required.
;Comp1 uses the insb instruction to insert the 3 bits of RLL code
;into an output buffer.
;The actual routine set-up is from label comp1 to label reline.
;        Registers used:
;                r0 = source base address
;                r1 = source starting bit offset
;                r2 = working run length
;                r3 = maximum run length
;                r4 = maximum bit offset (source)
;                r5 = address of destination buffer
;                r6 = bit offset of destination buffer
;                r7 = scan line count
comp1:
        addrd        mess,r0                ;point to base
        movqd        7,r3                ;max run length is 3 bits
        movd        32,r4                ;maximum bit offset
        addrd        buf,r5                ;destination buffer
        movqd        0,r6                ;starting bit offset
        movd        32,r7                ;line count
```

```
reline:        movqd        0,r1                ;starting bit offset
lp2:        TBITS        1                 ;look for a zero
        insb        r6,r2,0(r5),3        ;put them bits in
        addqd        3,r6                ;increment bit offset for
destination
        bhi        limit1                ;we hit the limit
lp3:        TBITS        0                 ;look for a one
        insb        r6,r2,0(r5),3        ;put them bits in
        addqd        3,r6                ;increment bit offset for
destination
        ble        lp2                ;if less than the limit, back for more
limit1:
        addqd        4,r0                ;next line now
        acbd        -1,r7,reline        ;decrement, and loop if more lines
        ret        0                ;amd we are done
```

```
;
;Comp2 does a fixed length RLL code with embedded 0/1 information.
;The run length is 5 bits, with one bit being the 0/1 flag.
;This gives a maximum run length of 15. We are compressing a
;32 by 32 font. Comp2 assumes that all lines begin with zeros.
;
;
;        Registers used:
;                r0 = source base address
;                r1 = source starting bit offset
;                r2 = working run length
;                r3 = maximum run length
;                r4 = maximum bit offset (source)
;                r5 = address of destination buffer
;                r6 = bit offset of destination buffer
;                r7 = scan line count
;
comp2:
        addrd        mess,r0                ;point to base
        movd        15,r3                ;max run length is 4 bits
        movd        32,r4                ;maximum bit offset
        addrd        buf,r5                ;destination buffer
        movqd        -5,r6                ;starting bit offset (we add 5)
        movd        32,r7                ;line count
reln2:        movqd        0,r1                ;starting bit offset
zero:        addqd        5,r6                ;incr. dest. bit offset
        TBITS        0                ;look for a one
        insb        r6,r2,0(r5),5        ;put them bits in
        bhi        limit2                ;if over limit, deal with it.
        bfc        zero                ;if still clear, back for more
one:        addqd        5,r6                ;increment dest. bit offset
        TBITS        1                ;look for a zero
        orb        h'10,r2                ;set the '1s' flag
        insb        r6,r2,0(r5),5        ;put them bits in
        bhi        limit2                ;if over limit, deal with it.
        bfs        one                ;if still one, look for a one
```

```
        br        zero                ;else look at zeros again

limit2:        addqd        4,r0                ;move to start of next line
        acbd        -1,r7,reln2        ;check and loop if more lines
;        addqd        5,r6                ;optional incr. of dest. bit. off.
        ret        0                ;else we are done
```

```
;
;Comp3 does a fixed length RLL code with embedded 0/1 information.
;Comp2 does not depend on the first part of a scan line begin zero, and
;auto-determines the optimal code. The resultant file can still be
;decompressed with UNCOMP2. This is a 32x32 font.
;
;
;        Registers used:
;                r0 = source base address
;                r1 = source starting bit offset
;                r2 = working run length
;                r3 = maximum run length
;                r4 = maximum bit offset (source)
;                r5 = address of destination buffer
;                r6 = bit offset of destination buffer
;                r7 = scan line count
;
comp3:
        addrd        mess,r0                ;point to base
        movd        15,r3                ;max run length is 4 bits
        movd        32,r4                ;maximum bit offset
        addrd        buf,r5                ;destination buffer
        movqd        -5,r6                ;dest. starting bit offset
        movd        32,r7                ;line count
reln3:        movqd        0,r1                ;starting bit offset
zero3:        addqd        5,r6                ;increment dest. bit offset
        TBITS        0                ;check for a one
        insb        r6,r2,0(r5),5        ;put them bits in
        bhi        limit3                ;if over limit, deal with it.
        bfc        zero3                ;if still clear, back for more
        cmpqd        0,r2                ;was rll code a zero?
        beq        one3a                ;yes! skip increment and overwrite 0
one3:        addqd        5,r6                ;increment dest. bit offset
one3a:        TBITS        1                ;look for a zero
        orb        h'10,r2                ;set the '1s' flag
        insb        r6,r2,0(r5),5        ;put them bits in
        bhi        limit3                ;if over limit, deal with it.
        bfs        one3                ;if still one, look for a one
        cmpqd        0,r2                ;was last rll code a zero?
        bne        zero3                ;no. look at zeros again
        addqd        -5,r6                ;it was. overwrite the 0
        br        zero3                ;and look at zeros again

limit3:        addqd        4,r0                ;move to start of next line
        acbd        -1,r7,reln3        ;check and loop if more lines
```

```
;        addqd        5,r6                ;optional incr. of dest. bit. off.
        ret        0                  ;else we are done
```

Rotate font emulation
ROTIMG

The new rotate image algorithm (developed by the Electronic Imaging Group
at National Semiconductor) does a very fast 8 by 8 (64 bit) rotation of a font data. Note also that this algorithm does not exclusively deal with fonts, but any 64 bit image.

ROTIMG (Rotate Image)

Inputs:
R0 = Source character address
R1 = Source character warp
R4 = Rotate table address

Outputs:
R2 = Destination character low 4 bytes (lsb->msb, 0 - 3)
R3 = Destination character high 4 bytes (lsb->msb, 4 - 7)

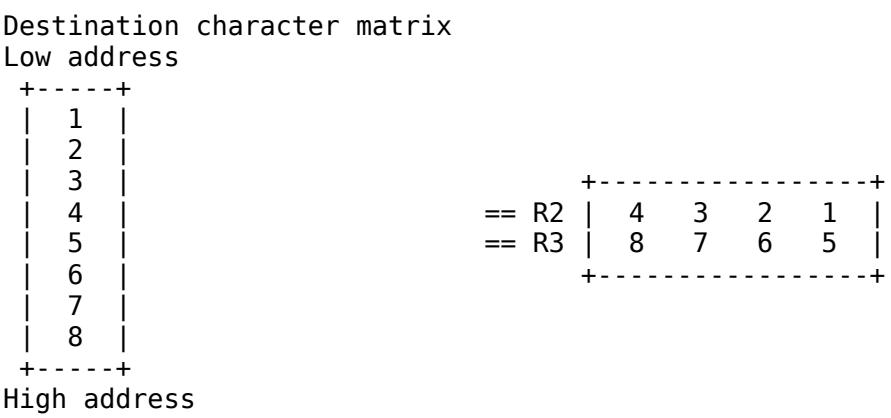
The source data is pointed to by R0 on entry. A 'source warp' is contained in R1. This 'source warp' is added to R0 after each read of the source data. This allows the rotation of 16 by 16, 32 by 32 and larger fonts. By way of an example, let's rotate an 8 x 8 character.

Character table for ">"

		Bit number							Hex Value
		0	1	2	3	4	5	6	7	
Byte	0	0	1	0	0	0	0	0	0	02
	1	0	0	1	0	0	0	0	0	04
	2	0	0	0	1	0	0	0	0	08
	3	0	0	0	0	1	0	0	0	10
	4	0	0	0	0	1	0	0	0	10
	5	0	0	0	1	0	0	0	0	08
	6	0	0	1	0	0	0	0	0	04
	7	0	1	0	0	0	0	0	0	02

ROTIMG deals with the 8 by 8 destination character as 8

sequential bytes in two registers (R2 and R3), as follows:



ROTIMG uses an external table (address in R4) to speed the rotation, and to limit the amount of code used. This table consists of 256 64 bit entries, or a total of 2,048 bytes. The table may be located code (PC) or data (SB) relative. The complete table is at the end of this document.

To rotate the character, the following example would be executed.

```
#
# Rotate Example
#
src:    .byte       0x2,0x4,0x8,0x10,0x10,0x8,0x4,0x2
dst:    .byte       0,0,0,0,0,0,0,0

rot:    addr   src,r0    #point to the source character
        movqd  1,r1      #the source warp is one
        addr   rottab,r4 #point to the ROTIMG table
        ROTIMG           #rotate the character
        movd   r2,dst    #store the dest. character low 32 bits
        movd   r3,(dst+4)#store the high 32 bits
#
# The character has been rotated and stored.
#
```

At this point, the data stored at 'dst:' is ready to be aligned, and imaged with one of the BITBLT (Bit Aligned Block Transfer) instructions. Below is the data as it is stored in 'dst:'.

Destination character table for ">", 90 degree

```
                Bit number                      Hex Value
           0  1  2  3  4  5  6  7
Byte   0  0  0  0  0  0  0  0  0                  00
       1  1  0  0  0  0  0  0  1                  81
       2  0  1  0  0  0  0  1  0                  42
       3  0  0  1  0  0  1  0  0                  24
       4  0  0  0  1  1  0  0  0                  18
       5  0  0  0  0  0  0  0  0                  00
       6  0  0  0  0  0  0  0  0                  00
       7  0  0  0  0  0  0  0  0                  00
```

Note that by re-ordering the output data, we may rotate 90 or 270 degrees. The following code, used in place of the two MOVD instructions in the previous example, rotates the character 270 degrees.

```
#
# Re-order table for 270 degree font rotation
#
     movd      r2,tos    #place low 32 bits on stack
     movd      r3,tos    #place high 32 bits on stack
     addr      dst,r0    #point to destination area
     movb      tos,3(r0) #and store destination data,
     movb      tos,2(r0) #one byte at a time
     movb      tos,1(r0)
     movb      tos,0(r0)
     movb      tos,7(r0)
     movb      tos,6(r0)
     movb      tos,5(r0)
     movb      tos,4(r0)
```

Destination character table for ">", 270 degree

```
                Bit number                      Hex Value
           0  1  2  3  4  5  6  7
Byte   0  0  0  0  0  0  0  0  0                  00
       1  0  0  0  0  0  0  0  0                  00
       2  0  0  0  0  0  0  0  0                  00
       3  0  0  0  1  1  0  0  0                  18
       4  0  0  1  0  0  1  0  0                  24
       5  0  1  0  0  0  0  1  0                  42
       6  1  0  0  0  0  0  0  1                  81
       7  0  0  0  0  0  0  0  0                  00
```

Timing

With unrolled 32000 code, the time for this algorithm is about 588 clocks. On the NS32CG16, this time is estimated to be 280 clocks.

Table 1

This table is used for the ROTIMG code. It is 256 entries of 64 bits each (8 bytes * 256 = 2048 bytes). There are two entries per line.

```
rottab1:
        double        h'00000000,h'00000000,h'00000001,h'00000000 ;0,1
        double        h'00000100,h'00000000,h'00000101,h'00000000 ;2,3
        double        h'00010000,h'00000000,h'00010001,h'00000000 ;4,5
        double        h'00010100,h'00000000,h'00010101,h'00000000 ;6,7
        double        h'01000000,h'00000000,h'01000001,h'00000000 ;...
        double        h'01000100,h'00000000,h'01000101,h'00000000
        double        h'01010000,h'00000000,h'01010001,h'00000000
        double        h'01010100,h'00000000,h'01010101,h'00000000
        double        h'00000000,h'00000001,h'00000001,h'00000001
        double        h'00000100,h'00000001,h'00000101,h'00000001
        double        h'00010000,h'00000001,h'00010001,h'00000001
        double        h'00010100,h'00000001,h'00010101,h'00000001
        double        h'01000000,h'00000001,h'01000001,h'00000001
        double        h'01000100,h'00000001,h'01000101,h'00000001
        double        h'01010000,h'00000001,h'01010001,h'00000001
        double        h'01010100,h'00000001,h'01010101,h'00000001
        double        h'00000000,h'00000100,h'00000001,h'00000100
        double        h'00000100,h'00000100,h'00000101,h'00000100
        double        h'00010000,h'00000100,h'00010001,h'00000100
        double        h'00010100,h'00000100,h'00010101,h'00000100
        double        h'01000000,h'00000100,h'01000001,h'00000100
        double        h'01000100,h'00000100,h'01000101,h'00000100
        double        h'01010000,h'00000100,h'01010001,h'00000100
        double        h'01010100,h'00000100,h'01010101,h'00000100
        double        h'00000000,h'00000101,h'00000001,h'00000101
        double        h'00000100,h'00000101,h'00000101,h'00000101
        double        h'00010000,h'00000101,h'00010001,h'00000101
        double        h'00010100,h'00000101,h'00010101,h'00000101
        double        h'01000000,h'00000101,h'01000001,h'00000101
        double        h'01000100,h'00000101,h'01000101,h'00000101
        double        h'01010000,h'00000101,h'01010001,h'00000101
        double        h'01010100,h'00000101,h'01010101,h'00000101
        double        h'00000000,h'00010000,h'00000001,h'00010000
        double        h'00000100,h'00010000,h'00000101,h'00010000
        double        h'00010000,h'00010000,h'00010001,h'00010000
        double        h'00010100,h'00010000,h'00010101,h'00010000
        double        h'01000000,h'00010000,h'01000001,h'00010000
        double        h'01000100,h'00010000,h'01000101,h'00010000
```

```
double        h'01010000,h'00010000,h'01010001,h'00010000
double        h'01010100,h'00010000,h'01010101,h'00010000
double        h'00000000,h'00010001,h'00000001,h'00010001
double        h'00000100,h'00010001,h'00000101,h'00010001
double        h'00010000,h'00010001,h'00010001,h'00010001
double        h'00010100,h'00010001,h'00010101,h'00010001
double        h'01000000,h'00010001,h'01000001,h'00010001
double        h'01000100,h'00010001,h'01000101,h'00010001
double        h'01010000,h'00010001,h'01010001,h'00010001
double        h'01010100,h'00010001,h'01010101,h'00010001
double        h'00000000,h'00010100,h'00000001,h'00010100
double        h'00000100,h'00010100,h'00000101,h'00010100
double        h'00010000,h'00010100,h'00010001,h'00010100
double        h'00010100,h'00010100,h'00010101,h'00010100
double        h'01000000,h'00010100,h'01000001,h'00010100
double        h'01000100,h'00010100,h'01000101,h'00010100
double        h'01010000,h'00010100,h'01010001,h'00010100
double        h'01010100,h'00010100,h'01010101,h'00010100
double        h'00000000,h'00010101,h'00000001,h'00010101
double        h'00000100,h'00010101,h'00000101,h'00010101
double        h'00010000,h'00010101,h'00010001,h'00010101
double        h'00010100,h'00010101,h'00010101,h'00010101
double        h'01000000,h'00010101,h'01000001,h'00010101
double        h'01000100,h'00010101,h'01000101,h'00010101
double        h'01010000,h'00010101,h'01010001,h'00010101
double        h'01010100,h'00010101,h'01010101,h'00010101
double        h'00000000,h'01000000,h'00000001,h'01000000
double        h'00000100,h'01000000,h'00000101,h'01000000
double        h'00010000,h'01000000,h'00010001,h'01000000
double        h'00010100,h'01000000,h'00010101,h'01000000
double        h'01000000,h'01000000,h'01000001,h'01000000
double        h'01000100,h'01000000,h'01000101,h'01000000
double        h'01010000,h'01000000,h'01010001,h'01000000
double        h'01010100,h'01000000,h'01010101,h'01000000
double        h'00000000,h'01000001,h'00000001,h'01000001
double        h'00000100,h'01000001,h'00000101,h'01000001
double        h'00010000,h'01000001,h'00010001,h'01000001
double        h'00010100,h'01000001,h'00010101,h'01000001
double        h'01000000,h'01000001,h'01000001,h'01000001
double        h'01000100,h'01000001,h'01000101,h'01000001
double        h'01010000,h'01000001,h'01010001,h'01000001
double        h'01010100,h'01000001,h'01010101,h'01000001
double        h'00000000,h'01000100,h'00000001,h'01000100
double        h'00000100,h'01000100,h'00000101,h'01000100
double        h'00010000,h'01000100,h'00010001,h'01000100
double        h'00010100,h'01000100,h'00010101,h'01000100
double        h'01000000,h'01000100,h'01000001,h'01000100
double        h'01000100,h'01000100,h'01000101,h'01000100
double        h'01010000,h'01000100,h'01010001,h'01000100
double        h'01010100,h'01000100,h'01010101,h'01000100
double        h'00000000,h'01000101,h'00000001,h'01000101
double        h'00000100,h'01000101,h'00000101,h'01000101
double        h'00010000,h'01000101,h'00010001,h'01000101
```

```
double        h'00010100,h'01000101,h'00010101,h'01000101
double        h'01000000,h'01000101,h'01000001,h'01000101
double        h'01000100,h'01000101,h'01000101,h'01000101
double        h'01010000,h'01000101,h'01010001,h'01000101
double        h'01010100,h'01000101,h'01010101,h'01000101
double        h'00000000,h'01010000,h'00000001,h'01010000
double        h'00000100,h'01010000,h'00000101,h'01010000
double        h'00010000,h'01010000,h'00010001,h'01010000
double        h'00010100,h'01010000,h'00010101,h'01010000
double        h'01000000,h'01010000,h'01000001,h'01010000
double        h'01000100,h'01010000,h'01000101,h'01010000
double        h'01010000,h'01010000,h'01010001,h'01010000
double        h'01010100,h'01010000,h'01010101,h'01010000
double        h'00000000,h'01010001,h'00000001,h'01010001
double        h'00000100,h'01010001,h'00000101,h'01010001
double        h'00010000,h'01010001,h'00010001,h'01010001
double        h'00010100,h'01010001,h'00010101,h'01010001
double        h'01000000,h'01010001,h'01000001,h'01010001
double        h'01000100,h'01010001,h'01000101,h'01010001
double        h'01010000,h'01010001,h'01010001,h'01010001
double        h'01010100,h'01010001,h'01010101,h'01010001
double        h'00000000,h'01010100,h'00000001,h'01010100
double        h'00000100,h'01010100,h'00000101,h'01010100
double        h'00010000,h'01010100,h'00010001,h'01010100
double        h'00010100,h'01010100,h'00010101,h'01010100
double        h'01000000,h'01010100,h'01000001,h'01010100
double        h'01000100,h'01010100,h'01000101,h'01010100
double        h'01010000,h'01010100,h'01010001,h'01010100
double        h'01010100,h'01010100,h'01010101,h'01010100
double        h'00000000,h'01010101,h'00000001,h'01010101
double        h'00000100,h'01010101,h'00000101,h'01010101
double        h'00010000,h'01010101,h'00010001,h'01010101
double        h'00010100,h'01010101,h'00010101,h'01010101
double        h'01000000,h'01010101,h'01000001,h'01010101
double        h'01000100,h'01010101,h'01000101,h'01010101 ;250,251
double        h'01010000,h'01010101,h'01010001,h'01010101 ;252,253
double         h'01010100,h'01010101,h'01010101,h'01010101 ;254,255
```